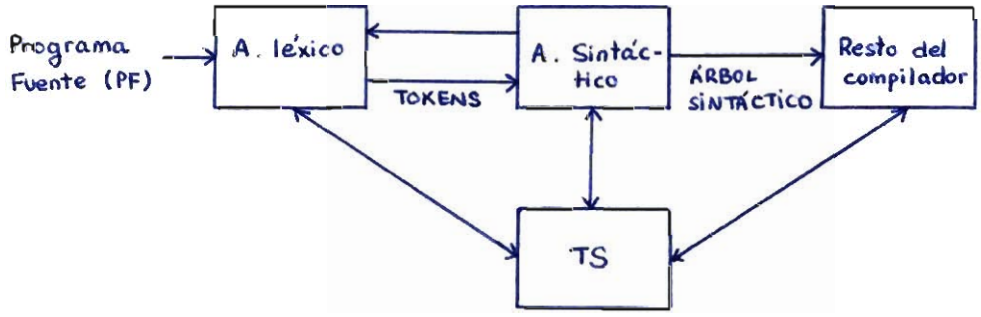
# TEMA 4: ANÁLISIS SINTÁCTICO.

TEMA 4.- Aspectos generales y algoritmos.
TEMA 5.- A. S. Ascendente: algoritmo LR y precedencia de operador.
TEMA 6.- A. S. Descendente: recursivo y LL (o basado en tablas).

## 1. INTRODUCCIÓN.

El analizador sintáctico es el motor del compilador: va recibiendo los tokens del a. léxico y cuando termina de tratar un token, entonces pide el siguiente al a. léxico.




El objetivo de este módulo es determinar si el programa fuente es correcto en función de la sintaxis del lenguaje, es decir, comprueba si el PF es correcto sintácticamente.

Para definir la sintaxis del lenguaje utilizo una definición formal y determinista de la misma. Esta definición se hace a través de una gramática. Si el PF se puede generar mediante dicha gramática será correcto; en caso contrario será incorrecto.

La gramática utilizada en el a. sintáctico debe ser de TIPO 2 (independientes de contexto):

$G = (N, T, P, S)$ ,, Producciones: $A \rightarrow \alpha$ / $A \in N$, $\alpha \in (N \cup T)^*$

Secuencia tokens $\begin{cases} \in L(G) \Leftrightarrow \text{Correcto} \\ \notin L(G) \Leftrightarrow \text{Incorrecto} \end{cases}$

El resultado que debe devolver el a. sintáctico como consecuencia del análisis de una secuencia es el **ÁRBOL SINTÁCTICO** correspondiente. Se trata de una secuencia de derivaciones que nos llevan a obtener la secuencia de tokens analizada.

Por tanto, analizar sintácticamente una secuencia de tokens consiste en encontrar para dicha secuencia un árbol sintáctico o árbol de derivación que, partiendo del axioma inicial de la gramática y a través de las reglas (producciones), obtengo en las hojas la secuencia de tokens analizada. Si esto se consigue se dice que la secuencia de tokens pertenece al lenguaje generado por la gramática, pudiendo generar el código correspondiente a la sentencia. En caso contrario, se genera el error adecuado a la situación y no se genera el código de esa sentencia. Es posible seguir analizando el resto del programa fuente para buscar más errores, pero ya no se generará ningún código.

La complejidad de generar el árbol de una sentencia de n tokens es $O(n^3)$. Es muy costoso porque tenemos en cuenta que la gramática puede ser cualquiera de tipo 2. Por ello, lo que se hace es elegir un subconjunto de este tipo de gramáticas de manera que podamos encontrar procedimientos que agilizan la generación del árbol, obteniendo así una complejidad menor a la anterior.

## * TIPOS DE MÉTODOS PARA GENERAR LOS A. SINTÁCTICOS:

- **UNIVERSALES:** Válidos para cualquier tipo de gramática de contexto libre. Complejidad $O(n^3)$.

- **DESCENDENTES (a izquierdas) Gramática LL.** (Top-down) Construyen el árbol partiendo del axioma de la gramática y aplicando reglas para generar las hojas, que serán la secuencia de tokens. Aplican derivación (siempre derivación a la izquierda).

- **ASCENDENTES (a derechas) Gramática LR.** (Bottom-up) Construyen el árbol partiendo de las hojas y aplicando las reglas de producción hasta llegar al axioma de la gramática. Aplican reducción-desplazamiento, no derivaciones por la derecha.

Los a. sintácticos ascendentes y descendentes son más sencillos que los universales. Imponen ciertas condiciones sobre la gramática, que se puede adaptar sin variar el lenguaje.

* **REPRESENTACIÓN DEL ÁRBOL SINTÁCTICO:**

- **ESTRUCTURA JERÁRQUICA**, donde cada nodo tendrá sus descendentes. El problema de esta notación es que no impone ningún orden cronológico de aplicación de las reglas.

- **SECUENCIA DE NÚMEROS** que representan el orden de aplicación de las reglas a partir del axioma para obtener el árbol correspondiente a la secuencia de tokens.

## 2. ANÁLISIS SINTÁCTICO DESCENDENTE.

Un "análisis a izquierdas" para una cadena terminal T es la lista de números de las reglas de derivación a izquierdas (reglas de producción) utilizadas para generar T a partir del axioma S de la gramática.

Una derivación a izquierdas consiste en aplicar una regla de producción al símbolo situado más a la izquierda.

> Un análisis sintáctico descendente de una sentencia consiste en tratar de encontrar un análisis a izquierdas para la cadena de entrada.

Si dada una gramática, al aplicar las reglas (producciones) obtengo una sentencia y al volver a aplicar las reglas en distinto orden obtengo la misma sentencia, entonces hay algo mal en la gramática. Para que el compilador funcione correctamente es necesario asegurar que para cada cadena de entrada se obtiene una ÚNICA salida. Se necesita una gramática (autómata) determinista porque al ir analizando la sentencia se van realizando también otras operaciones (acciones semánticas, generación de código, ...)

NO ADMITE GRAMÁTICAS AMBIGUAS.

Ejemplo: Análisis descendente.

Gramática tipo 2:

$G \equiv (N, T, P, E)$

$N = \{ E, T, F \}$

$T = \{ +, *, (, ), id \}$

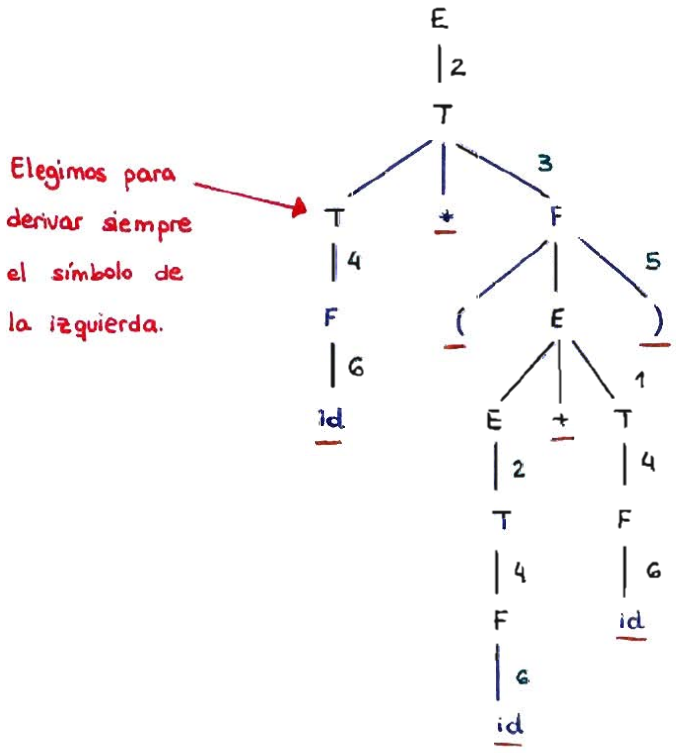
$P = \{$

| | |
|---|---|
| $E \rightarrow E + T$ | 1 |
| $E \rightarrow T$ | 2 |
| $T \rightarrow T * F$ | 3 |
| $T \rightarrow F$ | 4 |
| $F \rightarrow (E)$ | 5 |
| $F \rightarrow id$ | 6 |

$\}$

Esta gramática respeta (obliga a que se cumpla) la precedencia (prioridad) de los operadores aritméticos.

Sentencia a analizar: id * ( id + id)




ANÁLISIS DESCENDENTE (a izquierdas)

Orden de aplicación:

2 3 4 6 5 1 2 4 6 4 6.

Análisis descendente: derivar por la izquierda, del axioma a las hojas.

Es una sentencia válida porque obtengo un análisis correcto para ella.

Ejemplo: Ambigüedad.

- Producciones de la gramática: $E \to T + T \mid T * T$ 1 | 2
  $T \to E$ 3
  $T \to id$ 4

- Sentencia: a + b * c

Primera forma: $E \overset{1}{\Rightarrow} T + T \overset{4}{\Rightarrow} id + T \overset{3}{\Rightarrow} id + E \overset{2}{\Rightarrow} id + T * T \Rightarrow$
$id + id * T \underset{4}{\Rightarrow} id + id * id$

Segunda forma: $E \overset{2}{\Rightarrow} T * T \overset{3}{\Rightarrow} E * T \overset{1}{\Rightarrow} T + T * T \overset{4}{\Rightarrow} id + T * T \overset{4}{\Rightarrow}$
$id + id * T \underset{4}{\Rightarrow} id + id * id$

La gramática es ambigua porque las dos formas validan la sentencia. El compilador no acepta esta ambigüedad.

Necesitamos un proceso determinista. Para ello:
- Utilizamos una gramática no ambigua.
- Hacer que el analizador sea determinista. Por ejemplo, hacer que siempre haga derivaciones a izquierdas (que siempre coja el no terminal más a la izquierda). A veces esta solución no basta.
- No podemos tener una gramática recursiva por la izquierda.

Ejemplo: Recursividad por la izquierda.

$S \to Sb$
$S \to a$

Sentencia: w = ab

$S \Rightarrow Sb \Rightarrow Sbb \Rightarrow \dots \Rightarrow Sbbb..b$

Como siempre vamos eligiendo el primer no terminal por la izquierda, llegaríamos a una situación de bucle infinito y no se podría alcanzar nunca la validación de la sentencia.

Habría que eliminar la recursividad por la izquierda o hacer un análisis ascendente.

**GRAMÁTICA LL:** Con este tipo de gramática puedo construir un analizador sintáctico descendente que analiza la cadena de entrada de izquierda a derecha y produce un análisis a izquierdas correcto para dicha cadena de entrada.

## 3. ANÁLISIS SINTÁCTICO ASCENDENTE.

Un "análisis a derechas" para una cadena terminal T es la lista de números de las reglas de derivación derecha en orden inverso utilizada para generar la cadena T a partir del axioma S de la gramática.

Una derivación derecha consiste en aplicar una regla de producción al símbolo situado más a la derecha.

> Un análisis sintáctico ascendente de una secuencia consiste en tratar de encontrar un análisis a derechas (árbol a derechas) para la cadena de entrada.

Ejemplo: Análisis ascendente.

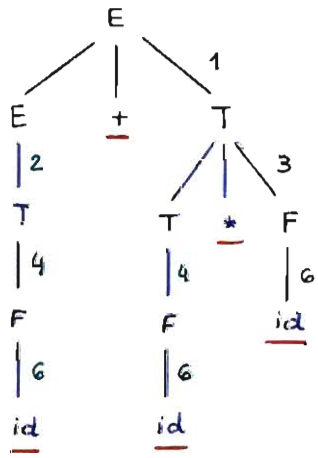
$G = (N, T, P, E)$

$P = \{$

| | | |
|---|---|---|
| $E \rightarrow E + T$ | 1 |
| $E \rightarrow T$ | 2 |
| $T \rightarrow T * F$ | 3 |
| $T \rightarrow F$ | 4 |
| $F \rightarrow (E)$ | 5 |
| $F \rightarrow id$ | 6 |

$\}$

Sentencia: id + id * id




Orden de aplicación:

6 4 2 6 4 6 3 1

Análisis ascendente: derivar por la derecha, del axioma a las hojas e invertir el orden de la lista.

Para comprobar la corrección del análisis ascendente anterior hacemos el "análisis a derechas descendente" y tiene que salir el mismo resultado, pero en orden inverso.

$$E \overset{1}{\Rightarrow} E + T \overset{3}{\Rightarrow} E + T * F \overset{6}{\Rightarrow} E + T * id \overset{4}{\Rightarrow}$$

$$E + F * id \overset{6}{\Rightarrow} E + id * id \overset{2}{\Rightarrow} T + id * id \overset{4}{\Rightarrow}$$

$$F + id * id \overset{6}{\Rightarrow} id + id * id$$

Orden de aplicación: 1 3 6 4 6 2 4 6

El orden de aplicación es inverso ⇒ El análisis ascendente obtenido es correcto.

**GRAMÁTICA LR:** Se caracteriza porque permite crear un analizador sintáctico ascendente analizando la cadena de entrada de izquierda a derecha, produciendo un análisis a derechas.

## 4. ANÁLISIS DESCENDENTE CON RETROCESO.

El retroceso de este algoritmo sirve para hacer determinista la gramática. Este método funciona para toda gramática de tipo 2 (de contexto libre) no recursiva por la izquierda, pues si lo fuera podría entrar en un bucle infinito.

Este algoritmo no es eficiente a causa del retroceso, por lo que no se suele utilizar para construir compiladores.

A continuación vamos a ver el algoritmo y un ejemplo.

→

PARSE = ANÁLISIS RESULTANTE = SECUENCIA DE LAS REGLAS APLICADAS.

## ANALIZADOR SINTÁCTICO DESCENDENTE CON RETROCESO

**Algoritmo:**

Entrada: Gramática de contexto libre no recursiva por la izquierda
Cadena de entrada: $w=a_1, a_2, \ldots a_n$, $n \geq 0$
Salida: Un *parse* a izquierdas para $w$, si existe; en otro caso, error

Procedimiento:

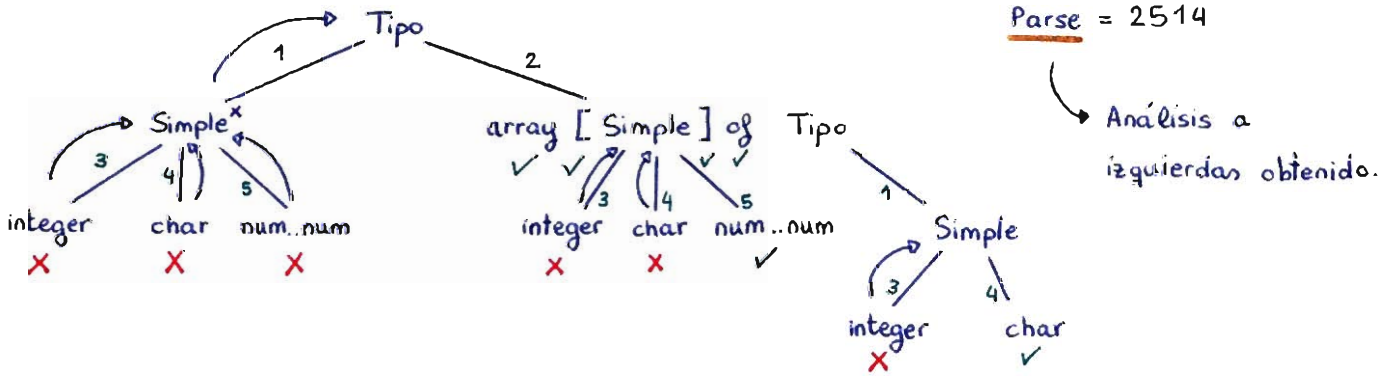
1. Ordenar y numerar todas y cada una de las reglas de la gramática
2. Comenzar la generación del árbol con el axioma de la gramática. Éste será el nodo activo, inicialmente
   Ejecutar:
   a. Si el nodo activo es un no terminal $A$:
      - Escoger la primera alternativa para $A$ y crear sus descendientes
      - Hacer el de más a la izquierda el nodo activo
      - Si $A$ no tiene más descendientes, hacer como nodo activo el siguiente a la derecha de $A$
   b. Si el nodo activo es un terminal $a$:
      - Comparar el *token* de entrada con $a$
        - Si son iguales, el nodo activo será el siguiente a la derecha de $a$ ✓
        - Si no lo son, regresar al nodo donde se aplicó la regla anterior previamente, y tratar la siguiente alternativa (otra derivación) x (Retroceso)
      - Si no hay más alternativas, volver al nodo anterior y seguir así x

   Hasta llegar al final de $w$ (si no se puede llegar a construir el árbol, error)
3. El *parse* será la secuencia de los números de las reglas utilizadas

**Ejemplo:** Algoritmo de análisis descendiente con retroceso.

$G \equiv (N, T, P, Tipo)$ $\quad w \equiv$ array [ num .. num ] of char

P: 1 Tipo → Simple
2 Tipo → array [ Simple ] of Tipo
3 Simple → integer
4 Simple → char
5 Simple → num .. num



# 5. ANÁLISIS ASCENDENTE CON RETROCESO.

Este método funciona para gramáticas sin ciclos y sin reglas λ (exceptuando, a veces, las reglas del axioma).

CICLO: $A \xrightarrow{+} A$ ≡ Vuelvo a obtener el símbolo no terminal A tras una serie de derivaciones.

Este algoritmo no se utiliza.

En general, los algoritmos con retroceso no son viables.

A continuación vamos a ver el algoritmo y un ejemplo.

## ANALIZADOR ASCENDENTE CON RETROCESO (REDUCCIÓN-DESPLAZAMIENTO)

### Algoritmo:

Entrada: Gramática de contexto libre sin reglas λ y sin ciclos
Cadena de entrada: $w=a_1, a_2, \ldots a_n$, $n \geq 1$
Las reglas de la gramática están numeradas
Salida: Un *parse* a derechas para *w*, si existe; en otro caso, error

Procedimiento:

1. Se considera que la cadena a analizar se encuentra en una pila de entrada
2. Explorar la cabeza de la pila de trabajo
   a. Comprobar si hay un lado derecho de alguna regla que pueda equipararse con los símbolos de la cabeza
      - Si es así:
        - Hacer una reducción, esto es, reemplazar estos símbolos por el lado izquierdo de la producción
        - Si puede hacerse más de una reducción, éstas estarán ordenadas de alguna forma para determinar cuál aplicar primero
      - Si no es posible la reducción:
        - Desplazar el siguiente *token* a la pila de trabajo y volver al paso 2 d.
   b. Si se llega al final de la cadena *w* y no ha sido posible una reducción, retroceder hasta el último movimiento en el cual se hizo una reducción. Si hay otra reducción posible en este punto, hacerla y volver al paso 2. Si no la hay, seguir retrocediendo
   c. Si se llega a un punto en que no se puede seguir avanzando, error
   d. Si se ha leído toda la cadena *w* y en la pila de trabajo sólo está el axioma, entonces el *parse* será la secuencia de los números de las reglas utilizadas en las reducciones

Pila de entrada ≠ Pila de trabajo. Son 2 pilas distintas:

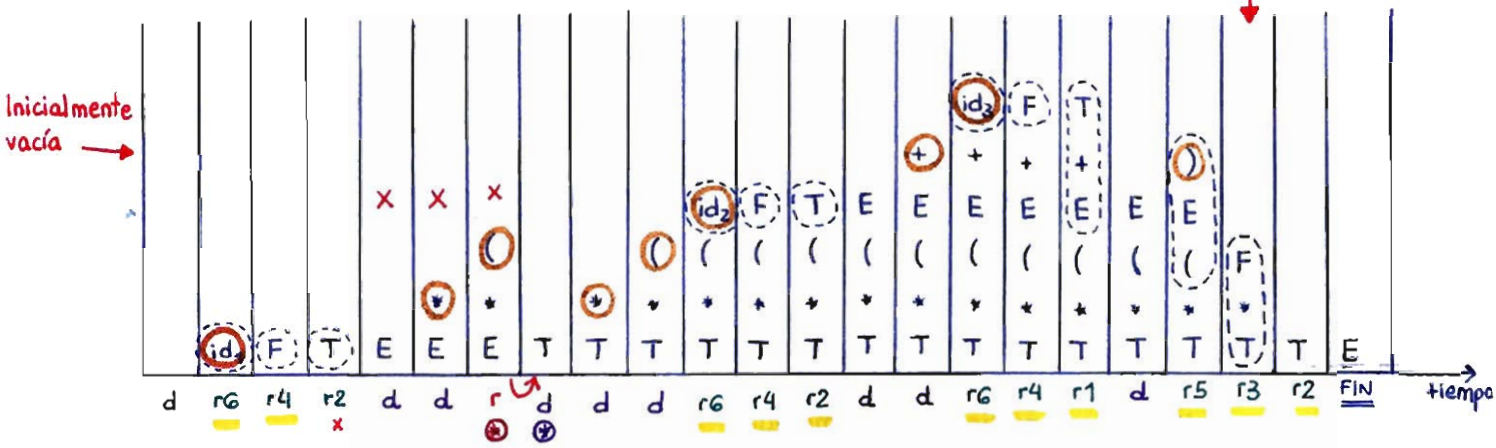
**Ejemplo:** Algoritmo de análisis ascendente con retroceso (Reducción - Desplazamiento).

P: 1 E → E + T
2 E → T
3 T → T * F
4 T → F
5 F → ( E )
6 F → id

Secuencia: $w \equiv id_1 * (id_2 + id_3)$

Se escoge r3 porque es la primera de las que se pueden aplicar.

Pila de trabajo:




(*) Por este camino llego al final de la secuencia y no obtengo solución válida (nunca encontraré E* en una parte derecha). Por ello retrocedo hasta la reducción r2.

(+) Desplazo porque no hay ninguna reducción posible, excepto r2, que ya hemos visto que no conduce a la solución.

**Parse** (análisis a derechas obtenido) = 6 4 6 4 2 6 4 1 5 3 2.

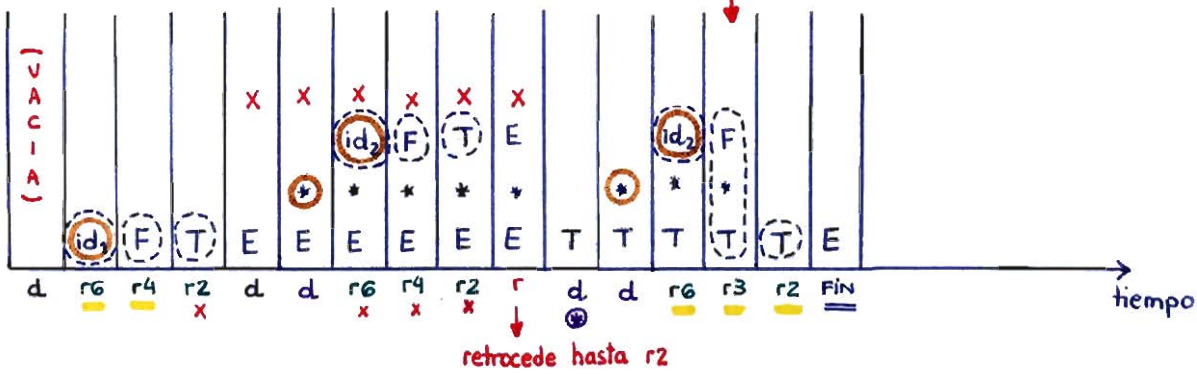
**Ejemplo:** Algoritmo de análisis ascendente con retroceso (Reducción - Desplazamiento).

P: 1 $E \rightarrow E + T$
2 $E \rightarrow T$
3 $T \rightarrow T * F$
4 $T \rightarrow F$
5 $F \rightarrow (E)$
6 $F \rightarrow id$

Secuencia: $w \equiv id_1 * id_2$

Se escoge r3 porque es la primera de las que se pueden aplicar.

Pila de trabajo:




(*) Desplazo porque no hay ninguna reducción posible, excepto r2, que ya hemos visto que no conduce a la solución.

**Parse** (análisis a derechas obtenido) = 6 4 6 3 2

## ANALIZADOR ASCENDENTE CON RETROCESO (REDUCCIÓN-DESPLAZAMIENTO)

**Algoritmo:**

Entrada: Gramática de contexto libre sin reglas $\lambda$ y sin ciclos
Cadena de entrada: $w=a_1, a_2,... a_n$, $n \geq 1$
Las reglas de la gramática están numeradas
Salida: Un *parse* a derechas para *w*, si existe; en otro caso, error

Procedimiento:

1. Se considera que la cadena a analizar se encuentra en una pila de entrada
2. Explorar la cabeza de la pila de trabajo
   a. Comprobar si hay un lado derecho de alguna regla que pueda equipararse con los símbolos de la cabeza
      - Si es así:
        - Hacer una reducción, esto es, reemplazar estos símbolos por el lado izquierdo de la producción
        - Si puede hacerse más de una reducción, éstas estarán ordenadas de alguna forma para determinar cuál aplicar primero
      - Si no es posible la reducción:
        - Desplazar el siguiente *token* a la pila de trabajo y volver al paso 2
   b. Si se llega al final de la cadena *w* y no ha sido posible una reducción, retroceder hasta el último movimiento en el cual se hizo una reducción. Si hay otra reducción posible en este punto, hacerla y volver al paso 2. Si no la hay, seguir retrocediendo
   c. Si se llega a un punto en que no se puede seguir avanzando, error
   d. Si se ha leído toda la cadena *w* y en la pila de trabajo sólo está el axioma, entonces el *parse* será la secuencia de los números de las reglas utilizadas en las reducciones

# ANALIZADOR SINTÁCTICO DESCENDENTE CON RETROCESO

**Algoritmo**:

Entrada: Gramática de contexto libre no recursiva por la izquierda
Cadena de entrada: $w=a_1, a_2,... a_n$, $n \geq 0$
Salida: Un *parse* a izquierdas para *w*, si existe; en otro caso, error

Procedimiento:
1. Ordenar y numerar todas y cada una de las reglas de la gramática
2. Comenzar la generación del árbol con el axioma de la gramática. Éste será el nodo activo, inicialmente
   Ejecutar:
   a. Si el nodo activo es un no terminal *A*:
      - Escoger la primera alternativa para *A* y crear sus descendientes
      - Hacer el de más a la izquierda el nodo activo
      - Si *A* no tiene más descendientes, hacer como nodo activo el siguiente a la derecha de *A*
   b. Si el nodo activo es un terminal *a*:
      - Comparar el *token* de entrada con *a*
        - Si son iguales, el nodo activo será el siguiente a la derecha de *a*
        - Si no lo son, regresar al nodo donde se aplicó la regla anterior previamente, y tratar la siguiente alternativa (otra derivación)
      - Si no hay más alternativas, volver al nodo anterior y seguir así

   Hasta llegar al final de *w* (si no se puede llegar a construir el árbol, error)
3. El *parse* será la secuencia de los números de las reglas utilizadas